Programming Assignment #1

Partitioning

* Graph partitioning is the reduction of a graph to a smaller graph by partitioning its set of nodes into mutually exclusive groups. Edges of the original graph that cross between the groups will produce edges in the partitioned graph.

* Applications in VLSI circuit design involve partitioning of a complex system into smaller subsystems is done hierarchically. It is done untill each subsystem has a manageable size. Each subsystem can then be designed independantly. The partitioning is done untill inuseterconnections between partitioins is minimized. This eases the task of interfacing the subsystems. This is also done because the communication between subsystems is usually costly.

Graph partitioning Methods

* There are two broad categories of methods, local and global.

* Local methods are the Kernighan-Lin algorithm, and Fiduccia-Mattheyses algorithms. Their major drawback is the arbitrary initial partitioning of the vertex set, which can affect the final solution quality.

* Global approaches include spectral partitioning, where a partition is derived from approximate eigenvectors of the adjacency matrix or spectral clustering that groups graph vertices using the eigendecomposition of the graph Laplacian matrix.

Iterative Improvement

The partitioining problem is the problem of breaking a circuit into two subcircuits which is solved by the method of Iterative improvement. To apply the iterative improvement technique, following needs to be done:

1. Generate an initial solution to the problem.

2. A criteria needs to be decided upon to determine whether or not a solution is acceptable.

3. An evaluation method needs to be determined to find the goodness of a solution.

4. Need one or more techniques for generating new solutions from existing solutions. The techniques must take the acceptability criteria into account so that only acceptable solutions are generated.

Iterative improvement algorithms start with an initial solution, and generate new solutions iteratively untill a solutioin that is as optimal as possible. Optimality is measured with respect to the goodness criteria. Most iterative improvement techniques do not produce optimal solutions, but can

be quite close to one. Most iteratie improvement techniques are greedy. When a new solution is generated from an existing solution, there are two choices: Discard the old solution and keep the new one, or discard the new solution and try some other technique for generating a new solution (or stop the process). In a greedy algorithm, the new solution is accepted only if it is better than the old one (with respect to the goodness criterion). Non-greedy methods (sometimes known as hill-climbing algorithms) will sometimes accept a solution that is worse than the existing solution. The

reason that hill-climbing algorithms are used is to avoid getting trapped in a local minimum. If solution A can be created from solution B in one step, then solution B is called a neighbor of solution A. It is possible for a non-optimal solution to be better than all of its neighbors. When this occurs, the solution is said to be a local minimum. A hill-climbing algorithm can sometimes climb out of a local minimum, and find an even better solution by temporarily accepting a solution that is worse than the existing solution.

Kernighan-Lin(KL) algorithm

The Kernighan-Lin algorithm is a heuristic algorithm for graph partitioining. To apply the algorithm to circuit partitioining the cicuit needs to be converted into a graph.

This can be achieved by treating each gate as a vertex of the graph. If two gates are directly connected by a net, then an edge is placed between the corresponding vertices of the graph. An approximation of a circuit partitioining can be obtained by partitioining the corresponding graph.

The KL algorithm works as follows:

1. The initial partition is generated “at random” i.e. create two subcircuits S1 and S2. If the circuit has n gates, the first n/2 are assigned to S1, and the rest are assigned to S2. Because the gates in a circuit description appear in what is essentially a random order, the initial partition appears to be random.

2. A solution is acceptable only if both subcircuits contain the same number of gates assuming the number of gates is even. The algorithm can be tweaked to handle an odd number of gates.

3. The goodness of a solution is equal to the number of graph edges that are cut. Suppose the edge (V, W) exists (where V and W are both gates) in the graph derived from the circuit. There are two possibilities. V and W can be in different subcircuits, or they can be in the same subcircuit. If V and W are in different subcircuits, the edge (V,W) is said to be *“cut”* . Otherwise the edge is *“uncut”*.

4. The technique for generating new solutions from old solutions is to select a subset of gates from S1, and a subset of gates from S2 and swap them. To maintain acceptability, two subsets of the same size are selected.

Selecting a subset of gates

The KL algorithm does not swap gates one at a time. Although, the algorithm does many single-gate swaps, these are all done on a temporary basis. The objective of doing these swaps is to

find a subset of gates to swap on a permanent basis.

The process of selecting a subset of gates involves the following steps:

The first step is to compute the cut and uncut counts for each vertex. For each vertex V in the circuit, two values C_V and U_v are calculated. C_V is the number of edges attached to V that are cut, while U_v is the number of edges attached to V that are uncut. The sum, $C_V + U_V$, is the total number of edges attached to V, which is also known as the degree of V.

Assume that K is the number of edges that are cut in the current partition. If vertex V is moved from one partition to the other, the change in K will be $I_v = C_v - U_V$ i.e. after moving V from one partition to the other, there will be new number of edges cut. Now, there will be $K – I_V$ cut edges in the circuit because, after moving V, any edge attached to V that was cut will no longer be cut, and any edge that was uncut will become cut. The number I_v is generally called the improvement with respect to V. A positive improvement is good, while a negatie improvement is bad.

The KL algorithm uses a series of pair-swaps to determine the set of gates to be swapped between partitions. Let V and W be vertices such that V is in one partition and W is in the other. The improvement $I_{(V,W)}$ that is obtained from swapping V and W is $I_v + I_W - 2$ if V and W are connected to one another and $I_V + I_W$ otherwise.

Once the combined improvement for all pairs of gates (there are $n^2 / 4$ such pairs) is computed, proceed with swapping pairs of gates. These swaps are tentative swaps, no actual movement of gates takes place. Before proceeding, compute the total number of cut nets in the circuit, and record that count as the “zero swaps” count.

Sort the list of gate pairs into descending sequence, and select the pair, (V,W), with the greatest improvement. Then perform a tentative swap of V and W. Then, reduce the total number of cut nets by $I_{(V,W)}$, and also update the improvement I_X for any gate attached to V or W. If the edge (V,X) was cut, reduce I_X by one. If the edge (V,X) was uncut, increase I_X by one. Also update the improvement $I_{(X,Y)}$ of any pair of gates containing X, and possibly reposition the pair (X,Y) in the list of sorted pairs of gates.

After completing these steps, mark V and W as having been swapped and delete any pair of gates containing either V or W from the list of sorted pairs of gates. Once a gate has been swapped with another gate, it won't be swapped a second time. Record the pair (V, W) along with the new total number of cut nets obtained after swapping V and W.

After completing these steps, go back to the list of sorted pairs of gates, and select the pair with the largest improvement, repeating the steps a second time. Continue selecting gate pairs until the list of available pairs is exhausted. This will cause every gate to be tentatively swapped with a gate in the other partition. If there are n gates in the circuit, there are now a lst of n/2 tentative swaps, along with a total cut count for each pair.

To determing the set of gates to be swapped, search the list of tentative swaps for the minimal total cut count. When finding the minimum, compare the nuber to the zero-swaps-count. If the minimum is less than the zero-swaps-count, then an actual swap is performed from the start of the list of tentative swaps up to, and including the pair that gave the smalled total cut count. If the smallest total cut count is greater than or equal to the zero-swaps-count, terminate the entire algorithm.

The steps above represent one iteration of the KL iterative improvement algorithm. After swapping a set of gates, the whole thing is done again, recomputing improvements, counting cut nets, and so forth. The entire algorithm stops when an iteration produes no improvement.

Implementation of KL algorithm

Main functions: In file include/read_data_helper.h

Reading data:

NodeKL* read_node_names(std::string nodes_file, unsigned int * numnodes, unsigned int * numterminals)

Inputs:

nodes_file: Path to nodes file

numnodes: Pointer to the number of nodes (including terminals)

numterminals: Pointer to the number of terminals

Output:

Returns a pointer to a list of nodes of type NodeKL

Function:

Reads the nodes_file and creates a list of nodes of type NodeKL and assigns each node its name read from the nodes_file. Then, returns the pointer to this list of nodes. Also, the function reads the number of nodes and the number of terminals from the nodes_file and populates the value of pointer numnodes and numterminals respectively (Since, they are passed by reference)

int read_nets(std::string nets_file, NodeKL* node_data, unsigned int num_term, unsigned int numnodes)

Inputs:

nets_file: Path to nets file

node_data: Pointer to the list of nodes of type NodeKL containing the name of the nodes

num_term: Number of terminals

numnodes: Number of nodes

Outputs:

Returns 0 if the reading of nets and storing the corresponding data was successful

Function:

Reads the nets file for connection between nodes and populates neighbors for each node as a pair of pointer to neighboring node and the corresponding distance. The distance between nodes is calculated as (1 / (Degree of net - 1)).

In file include/KL_helper.h:

Initial Random partition:

void initial_partition(unsigned int num_nodes, std::vector<NodeKL*> &group1, std::vector<NodeKL*> &group2, NodeKL * node_data)

Inputs:

num_nodes: Number of NodeKL nodes to be partitioned

&group1: Vector of pointers to NodeKL type object (Refers to I partition nodes)

&group2: Vector of pointers to NodeKL type object (Refers to II partition nodes)

node_data: Pointer to the array of nodes of type NodeKL containing the name of the nodes

Output:

Returns group1 and group2 vectors as passed by reference. group1 and group2 are pointers pointing to vector of pointers of NodeKL type. Also, updates the group member of NodeKL objects.

Function:

Generates the initial partitions as two vectors of pointers pointing to the nodes of NodeKL type. The partitions are generated by generating random numbers less the maximum number of nodes in the array (num_nodes). The pointer to nodes in the

array at the selected random numbers are placed in first vectors of NodeKL* type.

Calculating d values for initial random partition

void initial_d_values(unsigned int num_nodes, NodeKL * data)

Inputs:

num_nodes: Number of nodes

data: Pointer to the array of nodes of type NodeKL, each node containing its name, neighbors and the distance from the neighboring nodes stored as a pair.

Output:

Calculates and updates the d values for the nodes in the data array for the initial random partition

Function:

Calculates the D values for the initial random partition based on neighbors and the corresponding distance. The D values are calculated as follows:

Let A and B be the two partitions into which the nodes are to be divided.

$D(a) = E_a - I_a$ where E_a = External cost for node a = $\sum_{b \in B} c_{ab}$

I_a = Internal cost for node b = $\sum_{a \in A} c_{ab}$

c_{ab} = 0 if nodes a and b are not connected

= 1 if nodes a and b are connected by an edge

= (1 / Fanout) = (1 / (Number of connected nets – 1)) if nodes a and b are connected by a hyper-edge

Consider the nodes as shown in the following figure:

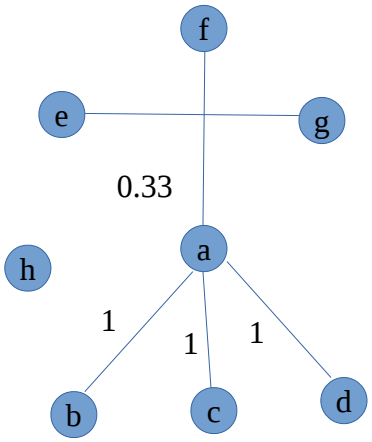


Fig. 1: Node “a” connected by edge/hyper-edge or no edge to other nodes.

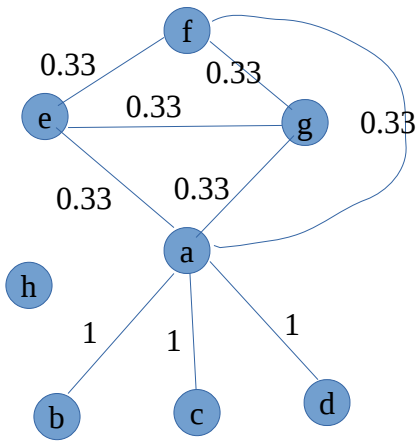


Fig. 2: Node “a” connected to hyper-edge nodes by an edge to every other node by weight = 0.33

Since, node a is not connected to node h, associated cost = 0

node a is connected to node b, node c and node d by an edge => Cost associated = 1

node a is connected to node e, f and g by a hyper-edge => Number of terminals connected = 4

=> Associated cost = 1 / (4 – 1) = 0.33

It is assumed that each node on a hyper-edge is connected to every other node on that hyper-edge by an edge with weight = 1 / (Nunmber of terminals - 1)

= 1 / (4 – 1) = 0.33 for the above figure (Fig. 2)

Calculating cut cost for a given partition

void cut_cost(unsigned int num_nodes, std::vector<NodeKL*> &group1, bool final = false)

Inputs:

num_nodes: Number of nodes

&group1: Pointer to Vector of pointers to NodeKL type object (Refers to I partition nodes)

*data: Pointer to the array of nodes of type NodeKL, each node containing its name, its group and its neighbors and the distance from the neighboring nodes stored as vector of pairs.

final: Boolean value which decides whether to print the final cut cost or an intermediate cut cost.

Outputs:

Displays cut cost for the given partitioining.

Function

Calculates and prints the cut cost of the current partitioining.

In file include/KL_graphHelper.h:

Setting the xy coordinates of the nodes

void set_xy_coordinates_for_nodes(unsigned int numnodes, NodeKL * data, int max_x, int max_y)

#Inputs:

numnodes: Number of nodes

*data: Pointer to the array of nodes of type NodeKL, each node containing its name, its group and its neighbors and the distance from the neighboring nodes stored as vector of pairs.

max_x: Window width on which to display graphical output

max_y: Window height on which to display graphical output

#Outputs:

Sets the x and y coordinates of the list of nodeKL objects for displaying on window.

Function:

Sets the x and y coordinates of the list of nodeKL objects for displaying on window. y coordinate is calculated as random number mod max_y. x coordinate is calculated as ((0 to 9) * max_x / 22) for first group and (max_x / 2 + (0 to 9) * max_x / 22) for second group to get five rows of nodes in each half of the window

Using SFML to open a window and draw the nodes and their connections

void drawGraph(unsigned int n_nodes, NodeKL * nodeList, int height, int width)

Inputs:

n_nodes: Number of nodes in the graph.

*nodeList: Pointer to the array of nodes of type NodeKL, each node containing its name, its group, x and y coordinates and its neighbors and the distance from the neighboring nodes stored as vector of pairs.

height: Window width on which to display graphical output

width: Window width on which to display graphical output

Outputs/Function:

Creates the “Graph” window. While the window is open, displays the node objects as circles with sf::CircleShape as the object type. The node objects are displayed according to their x and y coordinates set in set_xy_coordinates_for_nodes. Each node is colored in red with transperancy of 90. A line is drawn connecting to the

neighboring nodes with sf::RectangleShape as the object type. The list of neighboring nodes is read using the neighbors() function of each nodeKL object. The line color is based on the distance of the current node from the neighboring node (Red if distance > 0.67, Blue if distance > 0.33 and Green otherwise). A partition dividing line is drawn as sf::RectangleShape in blue color.

Drawing a line as a rectangle shape (sf::RectangleShape)

sf::RectangleShape Line(float x1, float y1, float x2, float y2, float thickness = 1)

Inputs:

x1,y1 and x2,y2 are the two end points of the line to be drawn. Thickness is the line thickness.

Outputs:

Returns an sf::RectangleShape type of object representing the line connecting the two end points

Function:

Creates a rectangle object with diagonally opposite points as the line end points The length for the line is given bylen = sqrt ($(x2 – x1)^2$ + $(y2 – y1)^2$) and angle of line is given by angle = $\tan^{-1}$((y2 – y1) / (x2 – x1)) * 180 / pi).
{ Rectange has length = len and width = thickness , fill color of (100,100,100) and rotation = angle }.

In file include/KL_bipartion_main.h

Main pass of the KL algorithm

void main_pass (unsigned int num_nodes, NodeKL * data, std::vector<NodeKL*> &A,

std::vector<NodeKL*> &B, int width, int height)

#Inputs:

num_nodes: Number of NodeKL nodes to be partitioned
*data: Pointer to the array of nodes of type NodeKL, each node containing its name, its group, x and y coordinates, its D value and its neighbors and the distance from the neighboring nodes stored as vector of pairs.
&A: Pointer to Vector of pointers to NodeKL type object (Refers to I partition of nodes)
&B: Pointer to Vector of pointers to NodeKL type object (Refers to II partition of nodes)
width: Window width on which to display graphical outputut
height: Window height on which to display graphical output

#Outputs:

Partitioning of nodes into two partitions group1/A and group2/B with minimum cut size.

#Function:

Partitions the nodes into vector of nodes such that optimal cut size is achieved by running the KL algorithm.

Determining the largest cut-cost reducing exchange nodes

void get_largest_decrease_cutcost_exchange_nodes (std::vector<NodeKL*> A, std::vector<NodeKL*> B, float * max_gain, NodeKL ** ELE1, NodeKL ** ELE2, unsigned int num_nodes)

#Inputs:

A: Pointer to Vector of pointers to NodeKL type object (Refers to I partition of nodes)

B: Pointer to Vector of pointers to NodeKL type object (Refers to II partition of nodes)

max_gain: Pointer to a float value passed by pointer. The maximum gain achieved by exchaning two nodes.

**ELE1: Pointer to pointer to NodeKL object present in the I partition

**ELE2: Pointer to pointer to NodeKL object present in the II partition

num_nodes: Number of NodeKL nodes to be partitioned

#Outputs:

ELE1 and ELE2 are the outputs passed by pointer along with the associated maximum gain achieved in reducing the cut cost.

#Function:

Determines the maximum gain achieved by calculating gain values associated with each pair of nodes that may be exchanged according to the following formula:

$$g_{ab} = D_a + D_b - 2 * (\text{Distance between nodes a and b})$$

For all the node pairs, the maximum gain calculated and the corresponding exchange nodes are returned by the function as pass by pointer values.

Deleting the locked max-gain exchange nodes from the temporary partitions used by main_pass

void delete_element(std::vector<NodeKL*> *group, NodeKL ele)

Inputs:

*group: Pointer to Vector of pointers to NodeKL type object from which element ele is to be deleted. Deletion here implies that the pointer pointing to ele object is removed from the vector of pointers.

ele: NodeKL type object element for which the pointer in the group vector is to be removed.

Outputs:

Updated *group vector with the pointer pointing to nodeKL object, in nodeList corresponding to ele, removed.

Function:

Deletes the pointer from vector of pointers in “group” pointing to nodeKL object in nodeList, corresponding to ele. Since, the group partition is passed by pointer, the changes are reflected in the calling function so that now group vector of pointers is decreased by one.

D value update as per the KL algorithm

int update_d_values2(unsigned int num_nodes, NodeKL * data, NodeKL* node1, NodeKL* node2)

Inputs:

num_nodes: Number of NodeKL type nodes to be partitioned

*data: Pointer to the array of nodes of type NodeKL, each node containing its name, its group, x and y coordinates, its D value and its neighbors and the distance from the neighboring nodes stored as vector of pairs.

*node1: Pointer to the NodeKL type object; The exchange node, part of the first partition, with maximum cut-cost reduction associated with it.

*node2: Pointer to the NodeKL type object; The exchange node, part of the second partition, with maximum cut-cost reduction associated with it.

Outputs:

D values of NodeKL objects in nodeList that are neighbors of the locked exchange nodes node1 and node2 are updated

Function:

As per the KL algorithm, the D values of the neighboring nodes to the maximum gain exchange nodes are updated by this function according to the following formula:

D_value update = D' + 2 * (Distance of same group locked exchange node) – 2 * (Distance of different group locked exchange node)

Since, D values are calculated based on distance, the D value update is also done based on the distance.

i.e. recalculate the D values for the elements of A - { a_i } and for the elements of B – {b_i }, by

$D_x' = D_x + 2c_{xai} - 2c_{xbi}$ $\quad x \in A - \{ a_i \}$

$D_y' = D_y + 2c_{ybi} - 2c_{ybi}$ $\quad y \in B - \{ b_i \}$

Maximising the partial sum of gains

int max_element_index(float * g, unsigned int max_index)

#Inputs:

g: Pointer to array of floats corresponding to the maximum gains achieved for each exchange nodes

max_index: Size of array g = numnodes / 2

Outputs:

Index with maximum cumulative gain.

Function:

Returns the index for which the cumulative max gain is the largest i.e. the partial sum of gains

Swapping elements actually after finding maximum gain index

void swap_ele(std::vector<NodeKL*> * G1, std::vector<NodeKL*> * G2, NodeKL *ele_A, NodeKL *ele_B)

#Inputs:

G1: Pointer to Vector of pointers to NodeKL type object from which element ele_A is to be removed and ele_B is to be added. Swap here implies that the pointer pointing to ele_A object is removed from the vector of pointers and the pointer pointing to ele_B object is added to the vector of pointers.

G2: Pointer to Vector of pointers to NodeKL type object from which element ele_B is to be removed and ele_A is to be added. Swap here implies that the pointer pointing to ele_B object is removed from the vector of pointers and the pointer pointing to ele_A object is added to the vector of pointers.

ele_A: Pointer to NodeKL type object part of partition 1 to be swapped

ele_B: Pointer to NodeKL type object part of partition 2 to be swapped

Outputs:

Updated G1 and G2 with swapped nodes.

Function:

Swap nodes corresponding to ele_A and ele_B from partition 1 and partition 2.

Data Structure Used:

The fundamental unit for the KL algorithm in this implementation is the NodeKL type object.

NodeKL object

```
class NodeKL: public Node {
    public:
        NodeKL(float x, float y, std::string name = ""): Node(x, y, name) {}
        NodeKL() {}
        void setDvalue(float D);
        float getDvalue();
        std::vector<std::pair<NodeKL*,float>> neighbors();
        void addNeighbor(NodeKL* node, float distance, bool distance_update);
        void update_distance (NodeKL* node, float distance);
        int getGroup();
        void setGroup(int g);
        float is_node2_connected(NodeKL node2);
    private:
        float _Dvalue;
        int _Group;
        std::vector<std::pair<NodeKL*,float>> _neighbors;
};
```

NodeKL class object is polymorphised on class Node.

=> It has following public member functions:

* Two NodeKL constructors which are overloaded. One accepts arguments as x and y coordinates and the object name. The other does not accept any arguments

* void setDvalue(float D): Function to set the _Dvalue private member of the class object.

* float getDvalue() : Function to retrieve the _Dvalue private member of the class object.

* std::vector<std::pair<NodeKL*, float>> neighbors() : Function to retrieve the vector of pairs of neighboring nodes and their corresponding distance.

* void addNeighbor(NodeKL * node, float distance, bool distance_update) : addNeighbor function is used to update the _neighbors private member of the NodeKL class with (neighboring node, distance) pairs.

* void update_distance (NodeKL *node, float distance) : update_distance function is used to add the distance to a node for a repeated connection. This function updates the distance for the vector pair in _neighbors for which node = node.

* int getGroup(): Returns the partition number of the NodeKL object. The returned number is an integer and has the value 1 or 2.

* void setGroup(int g): Sets the _Group private member of the NodeKL object which depicts the partition number for the node.

* float is_node2_connected(NodeKL node2): Returns the distance of the current node from the passed node “node2”.

=> It has the following private members:

float _Dvalue : Stores the D value of the node.
int _Group: Stores the partition number of the node
std::vector<std::pair<NodeKL*,float>> _neighbors : Stores the neighboring nodes and corresponding distances as vector of pairs of pointer to neighboring object and corresponding float distance.

Node object:

```
class Node {
  public:
    Node();
    Node(float x, float y, std::string name = "");
    Node(Node* node);
    float x();
    float y();
    std::string name();
    void setX(float x);
    void setY(float y);
    void setName(std::string name);
  private:
    float _x;
    float _y;
    std::string _name;
};
```

Class Node has following public member functions:

* It has the following three constructors that are overloaded. First constructor does not accept any arguments. Second accepts float type x, y coordinates and string type node object name initialized to empty string. Third accepts another node object to initialize the current node object.

* float x() : Retrieves the _x private member of the node.
* float y() : Retrieves the _y private member of the node.
* std::string name() : Retrieves the _name private member of the object
* void setX(float x) : Sets the _x private member of the node.
* void setY(float y) : Sets the _y private member of the node.
* void setName(std::string name) : Sets the _name private member of the objective

Class Node has the following private member functions:

* float _x : Stores the x coordinate of the node. This is used as the x-coordinate of window where

the node is drawn.
* float_y : Stores the y coordinate of the node. This is used as the y-coordinate of window where the node is drawn.
* std::string _name : Stores the name of the current object.

Results for experimental runs

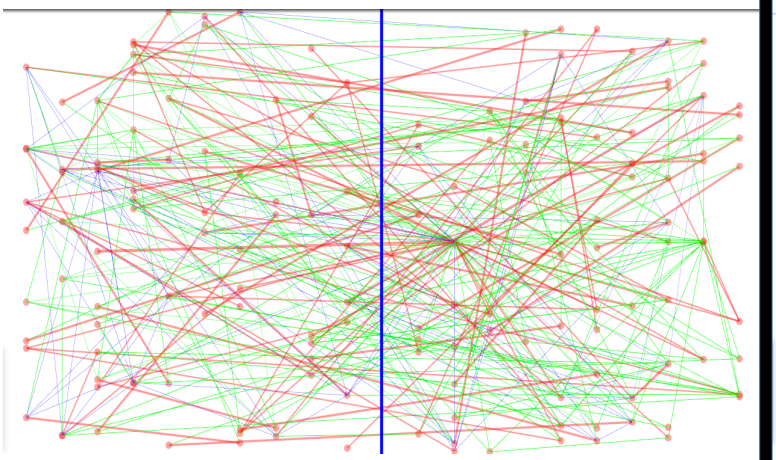
Following results were obtained with the KL partitioning algorithm implementation:

Benchmark circuit 1: ***151 nodes 167 edges***

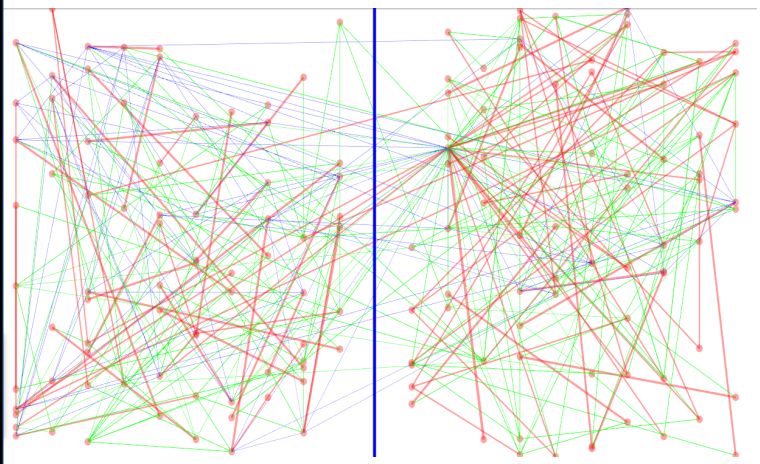
Files: spp_N151_E167_R11_80_nodes.txt
spp_N151_E167_R11_80_nets.txt

EXPERIMENT NUMBER : 1

Initial random partition:



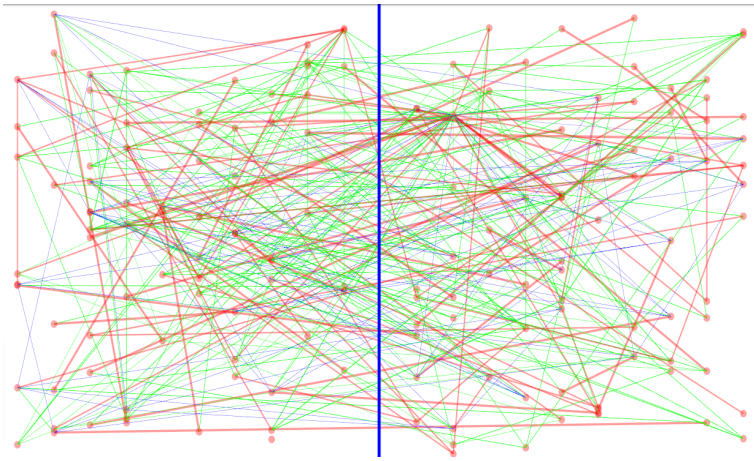
Final partitioining:



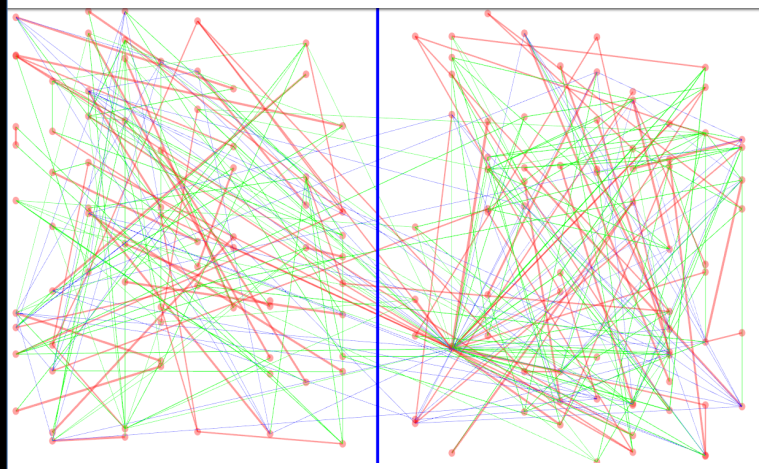
Initial Cut Size	Final Cut Size	Number of passes
109.74	20.89	7

EXPERIMENT NUMBER : 2

Initial random partition:



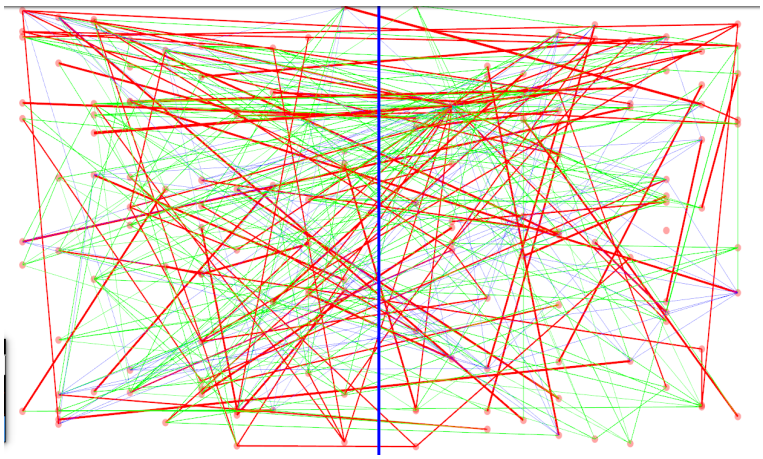
Final partitioining:



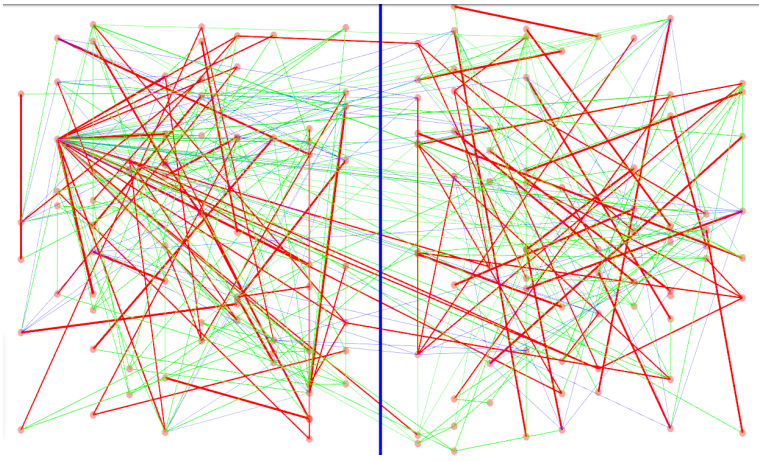
Initial Cut Size	Final Cut Size	Number of passes
107.83	23.44	4

EXPERIMENT NUMBER : 3

Initial random partition:



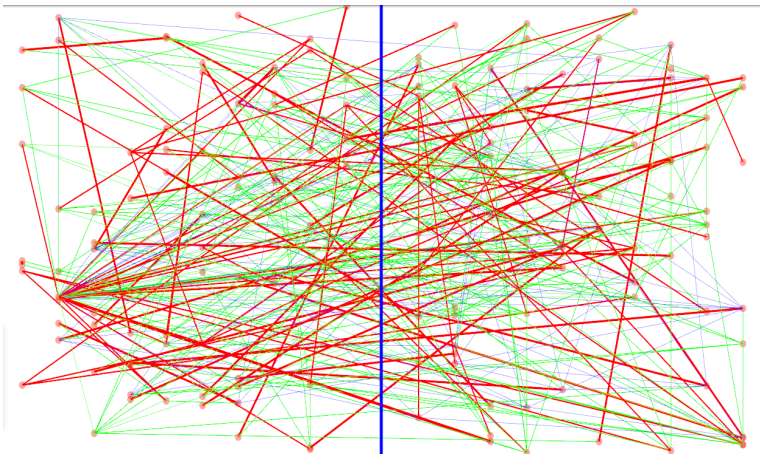
Final partitioning:



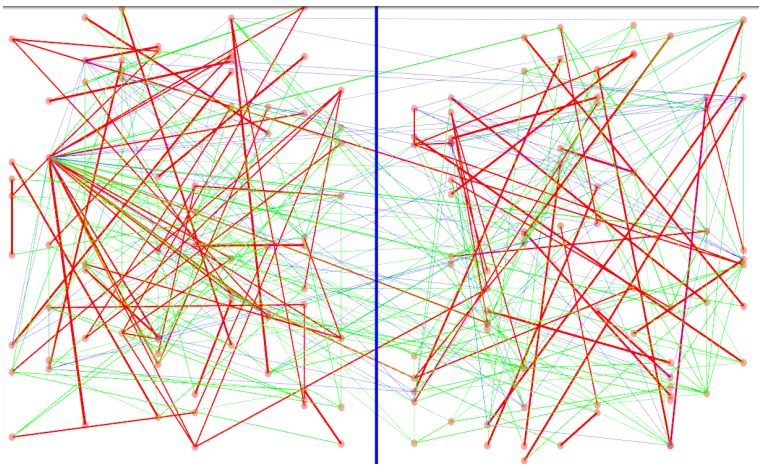
Initial Cut Size	Final Cut Size	Number of passes
112.63	27.57	5

EXPERIMENT NUMBER : 4

Initial Random partition:



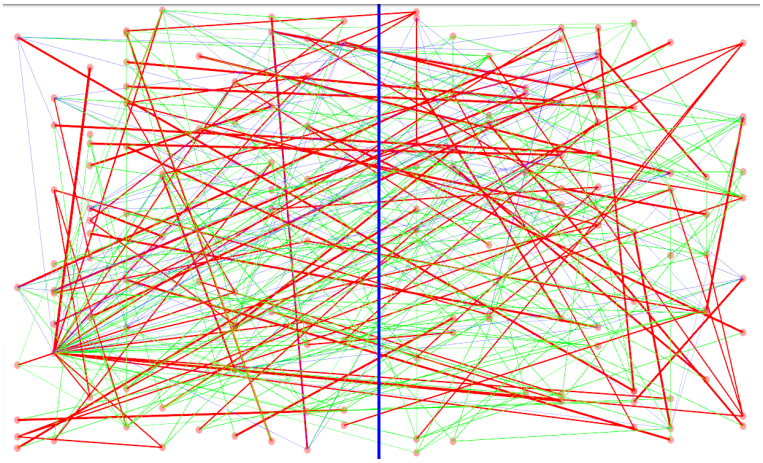
Final partitioining:



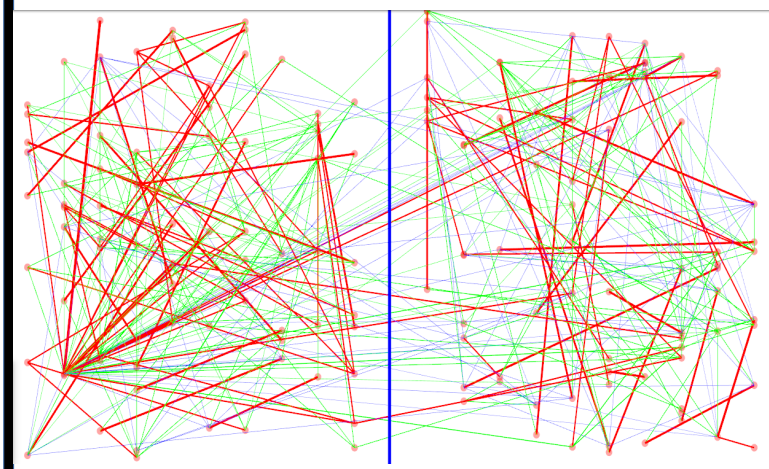
Initial Cut Size	Final Cut Size	Number of passes
118.12	27.17	3

EXPERIMENT NUMBER : 5

Initial random partition:



Final partitioining:



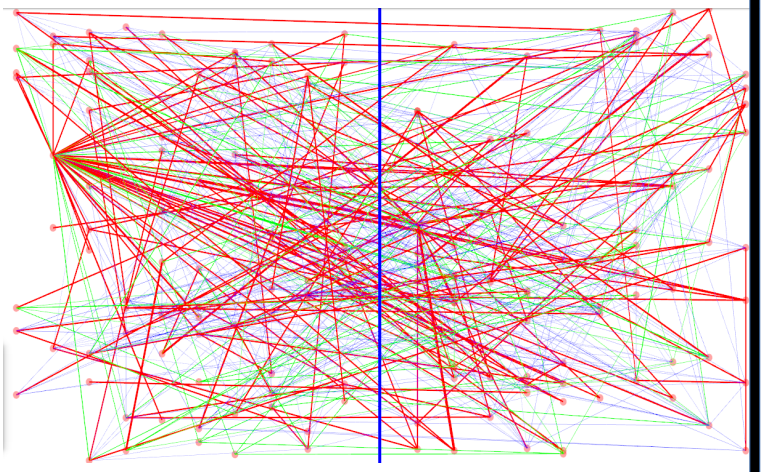
Initial Cut Size	Final Cut Size	Number of passes
104.84	22.36	7

Benchmark circuit 2: ***151 nodes 192 edges***

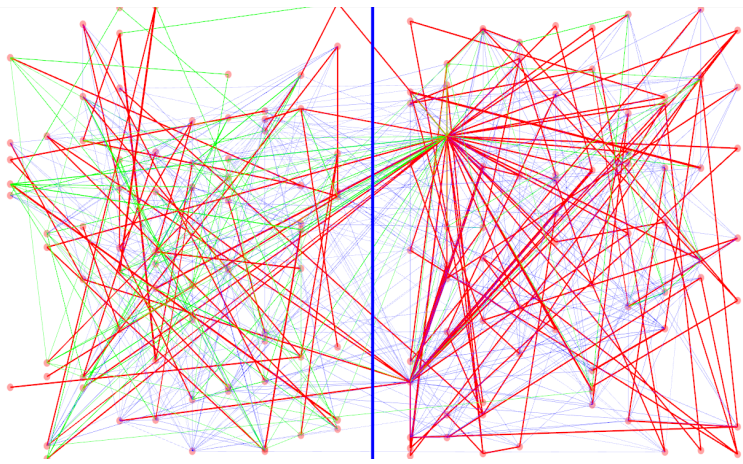
Files: spp_N151_E192_R8_232.nodes.txt
spp_N151_E192_R8_232.nets.txt

EXPERIMENT NUMBER : 1

Initial random partition:



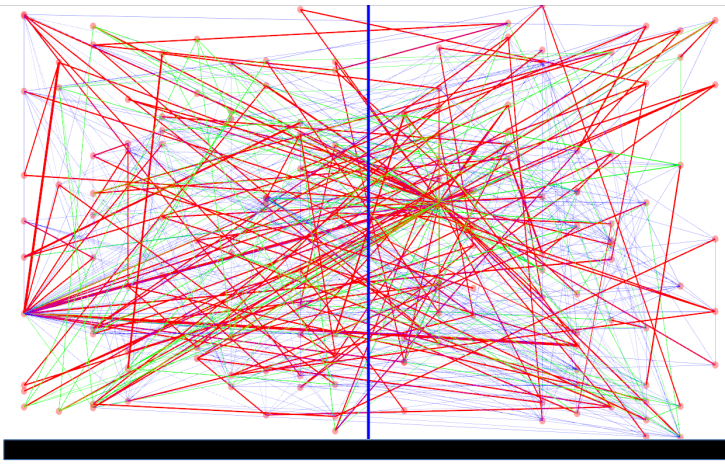
Final partitioining:



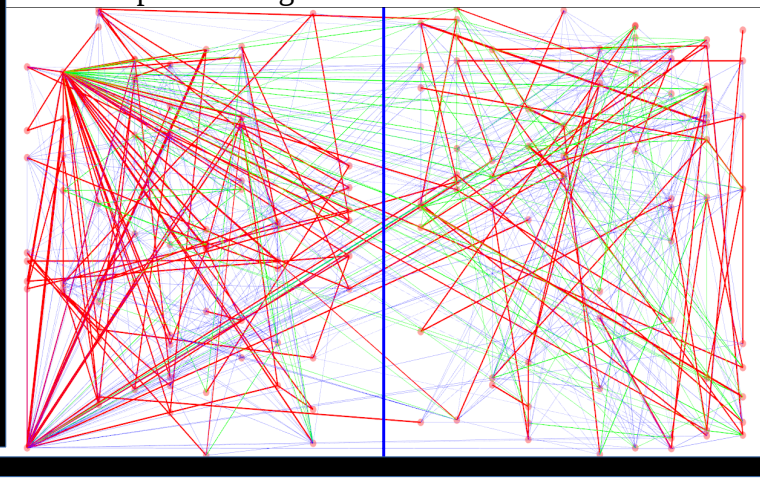
Initial Cut Size	Final Cut Size	Number of passes
131.66	33.82	4

EXPERIMENT NUMBER : 2

Initial random partition:



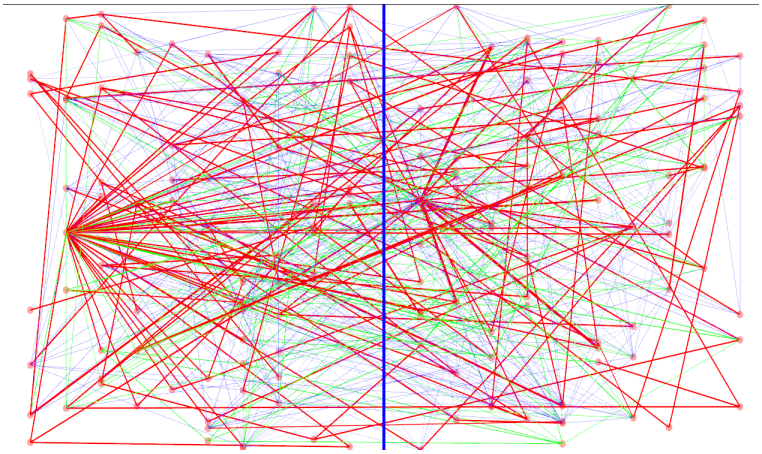
Final partitioining:



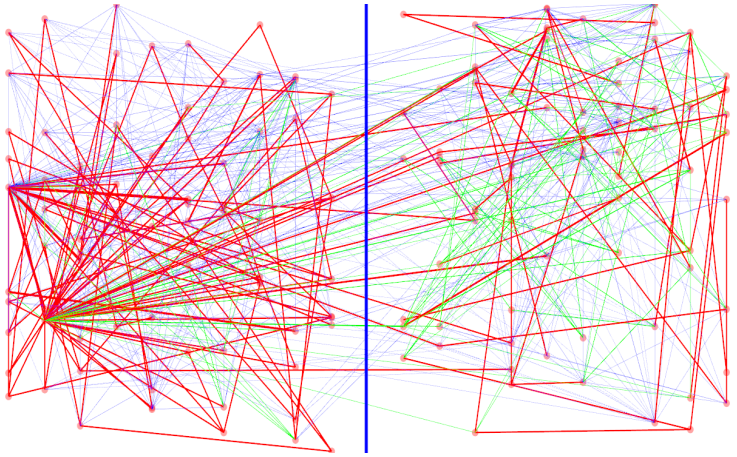
Initial Cut Size	Final Cut Size	Number of passes
136.12	36.13	5

EXPERIMENT NUMBER : 3

Initial random partition:



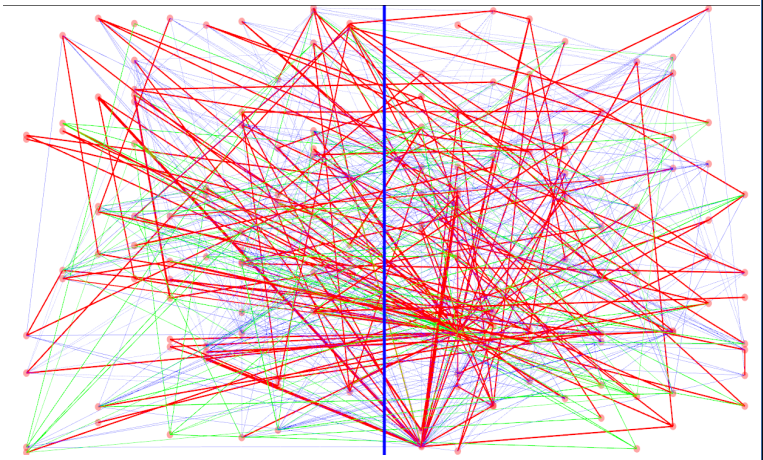
Final partitioining:



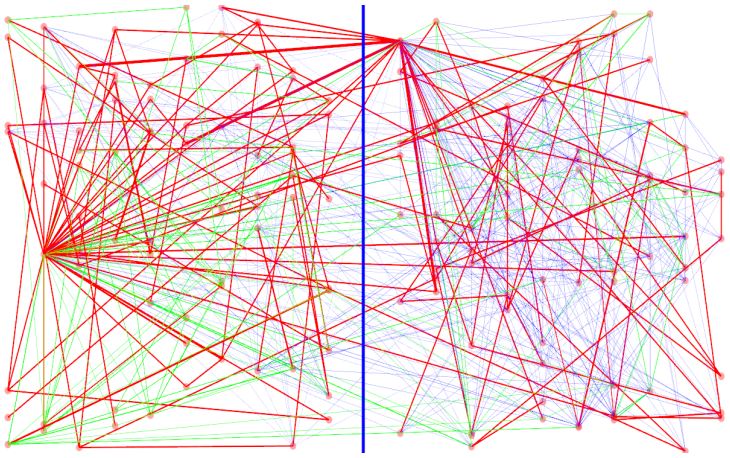
Initial Cut Size	Final Cut Size	Number of passes
116.43	34.82	5

EXPERIMENT NUMBER : 4

Initial random partition:



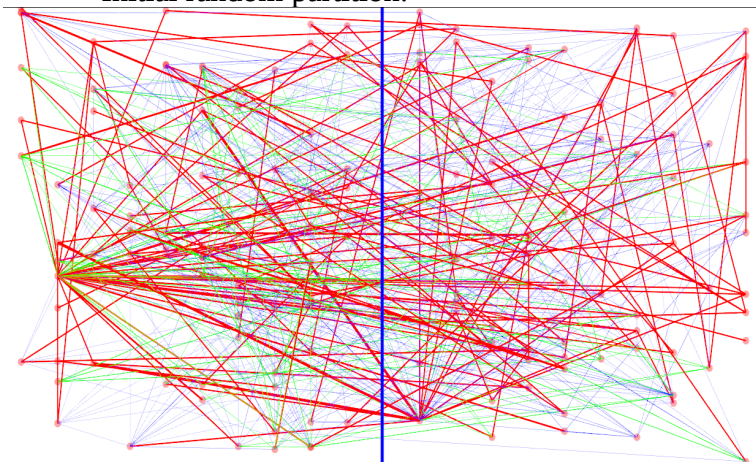
Final partitioining:



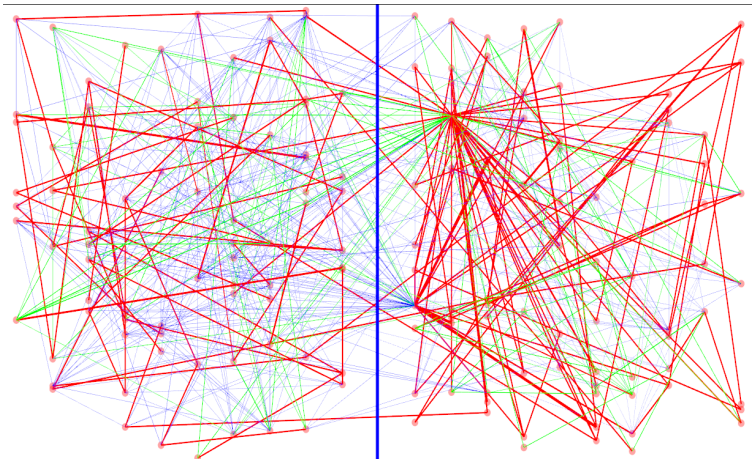
Initial Cut Size	Final Cut Size	Number of passes
136.6	42.74	5

EXPERIMENT NUMBER : 5

Initial random partition:



Final partitioning:



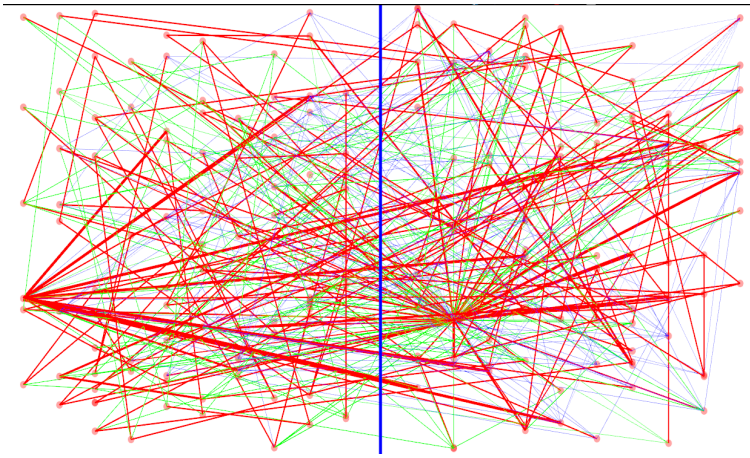
Initial Cut Size	Final Cut Size	Number of passes
120.9	38.01	3

Benchmark circuit 3: ***179 nodes 225 edges***

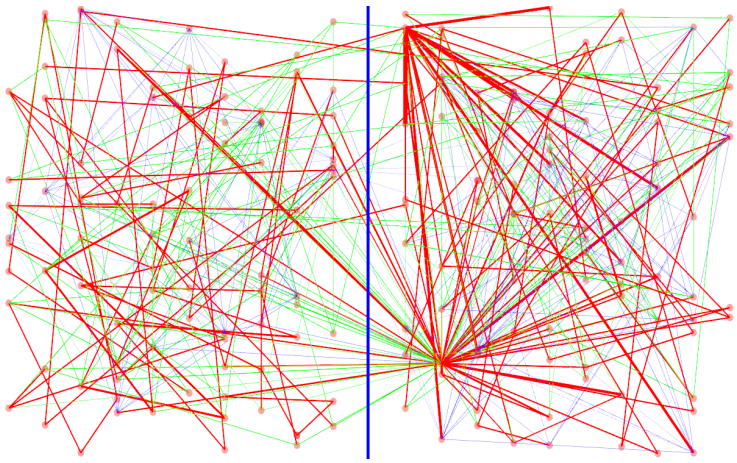
Files: spp_N179_E225_R11_158.nodes.txt
spp_N179_E225_R11_158.nets.txt

EXPERIMENT NUMBER : 1

Initial random partition:



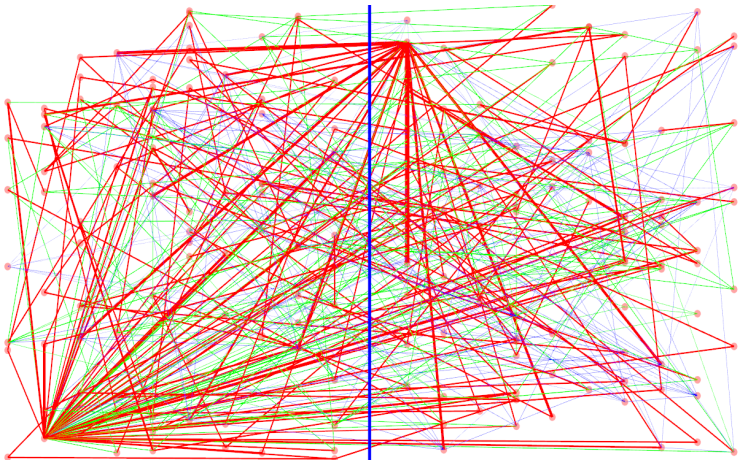
Final partitioining:



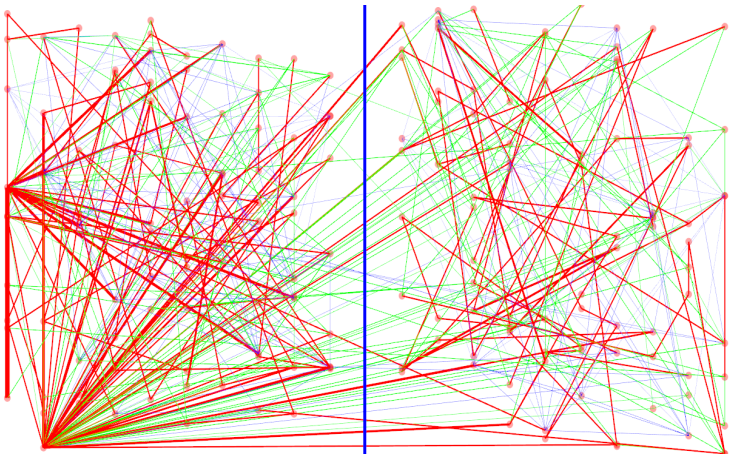
Initial Cut Size	Final Cut Size	Number of passes
127.27	35.91	4

EXPERIMENT NUMBER : 2

Initial random partition:



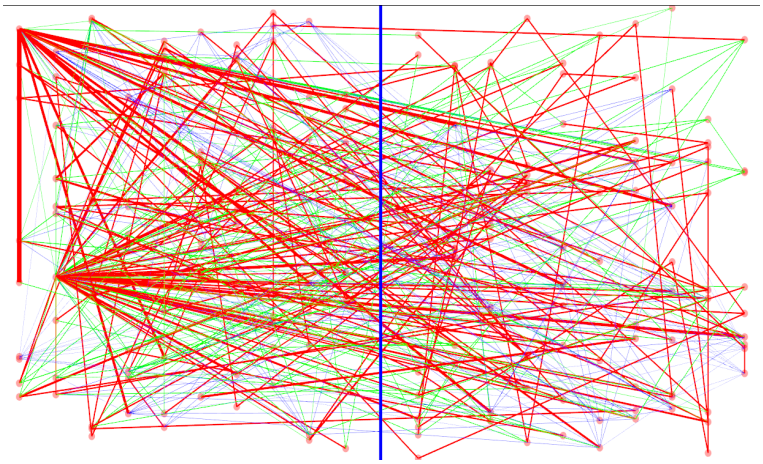
Final partitioining:



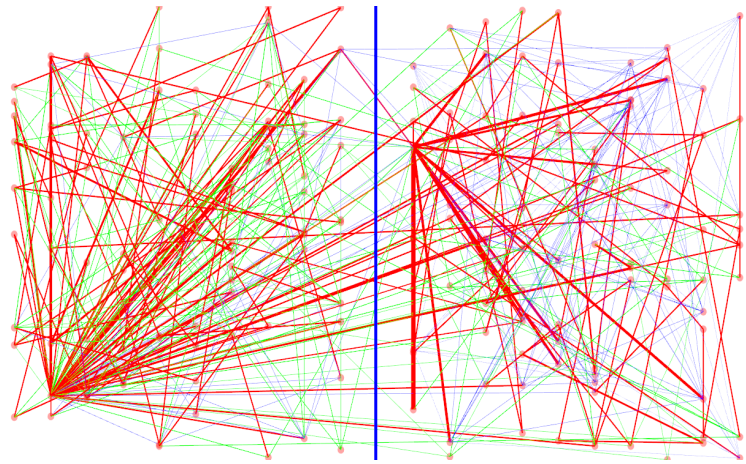
Initial Cut Size	Final Cut Size	Number of passes
134.56	34.39	5

EXPERIMENT NUMBER : 3

Initial random partition:



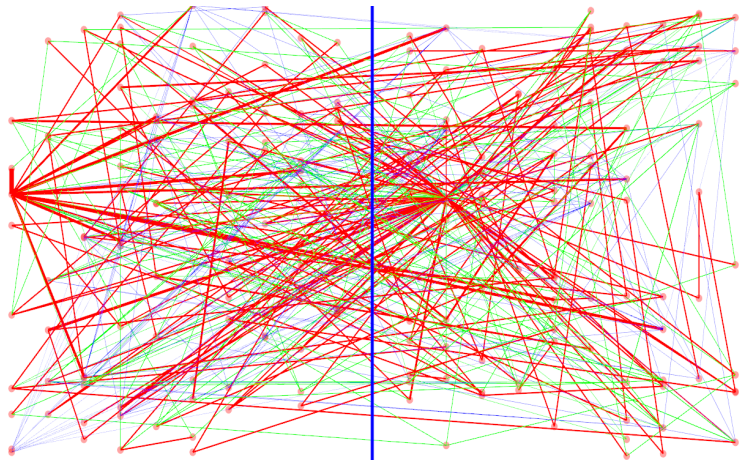
Final partitioining:



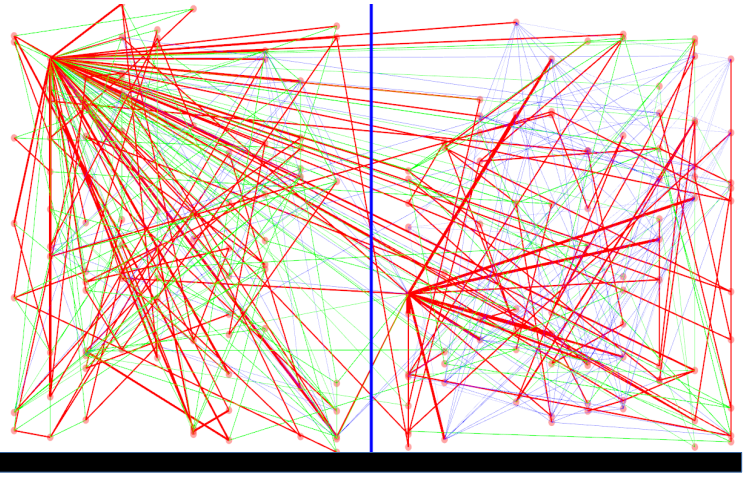
Initial Cut Size	Final Cut Size	Number of passes
146.63	37.43	4

EXPERIMENT NUMBER : 4

Initial random partition:



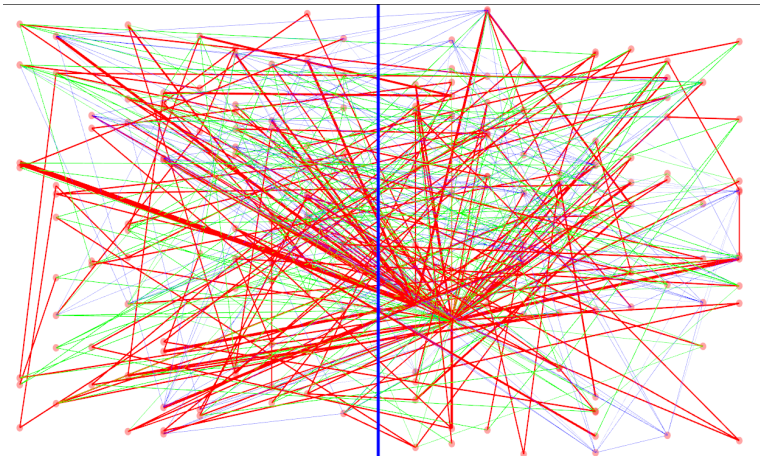
Final partitioining:



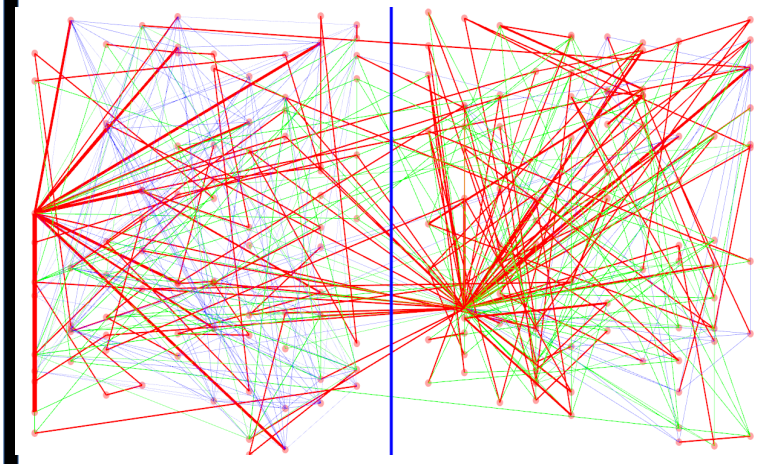
Initial Cut Size	Final Cut Size	Number of passes
143.39	38.27	7

EXPERIMENT NUMBER : 5

Initial random partition:



Final partitioining:



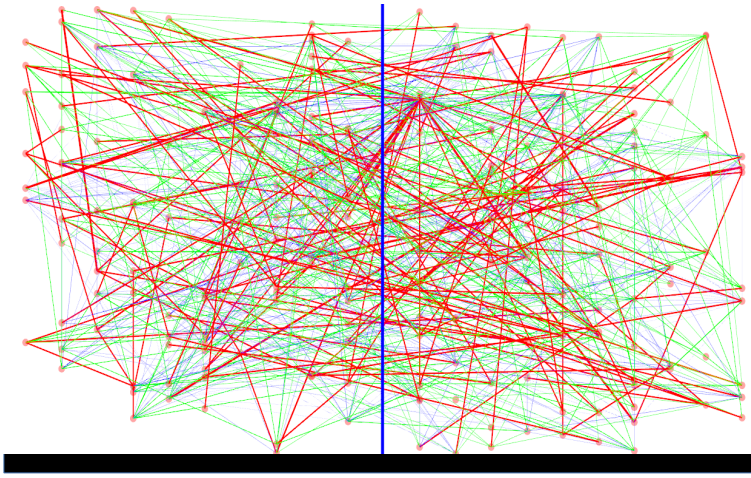
Initial Cut Size	Final Cut Size	Number of passes
147.85	38.9	4

Benchmark circuit 4: *189 nodes 227 edges*

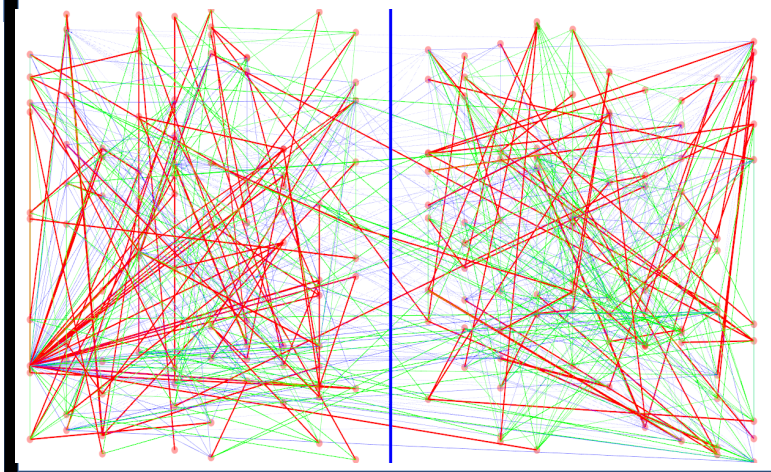
File : spp_N189_E227_R6_229.nodes.txt
spp_N189_E227_R6_229.nets.txt

EXPERIMENT NUMBER : 1

Initial random partition:



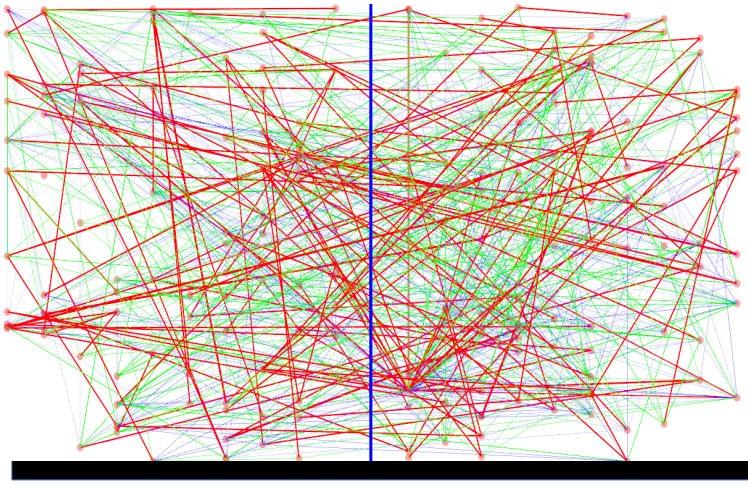
Final partitioining:



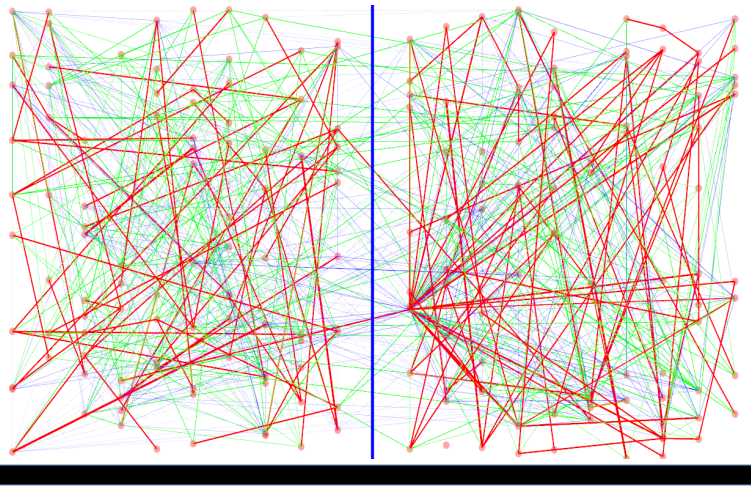
Initial Cut Size	Final Cut Size	Number of passes
155.75	32.12	7

EXPERIMENT NUMBER : 2

Initial random partition:



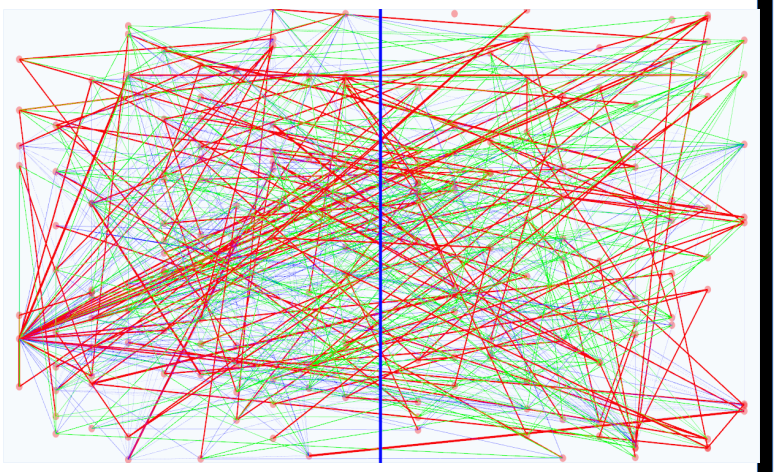
Final partitioining:



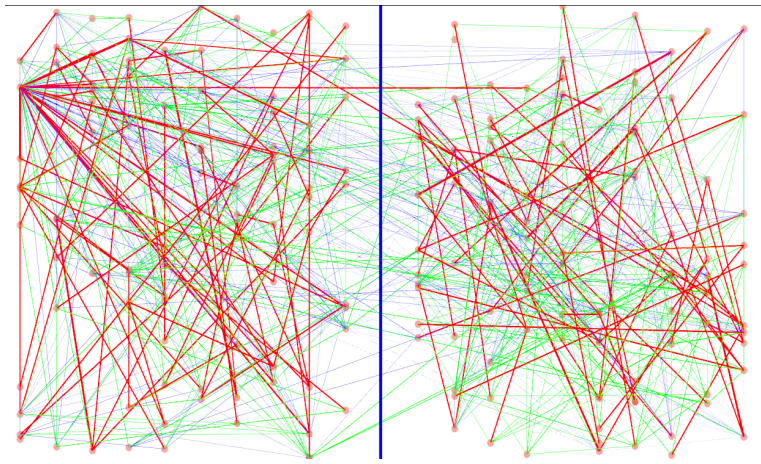
Initial Cut Size	Final Cut Size	Number of passes
152.14	30.58	7

EXPERIMENT NUMBER : 3

Initial random partition:



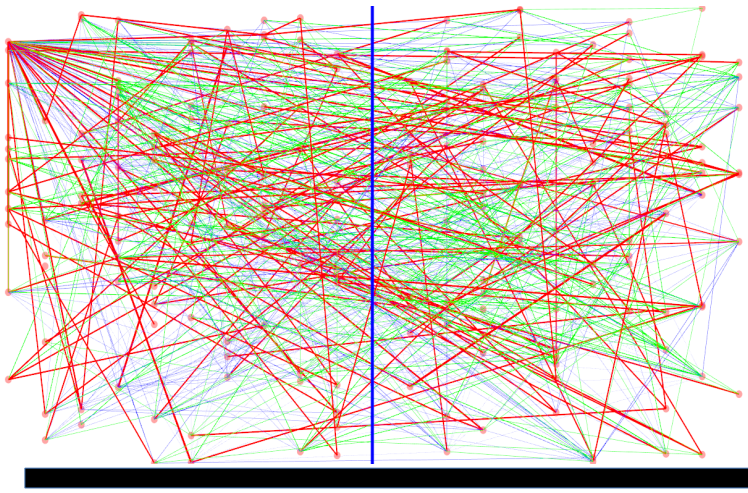
Final partitioining:



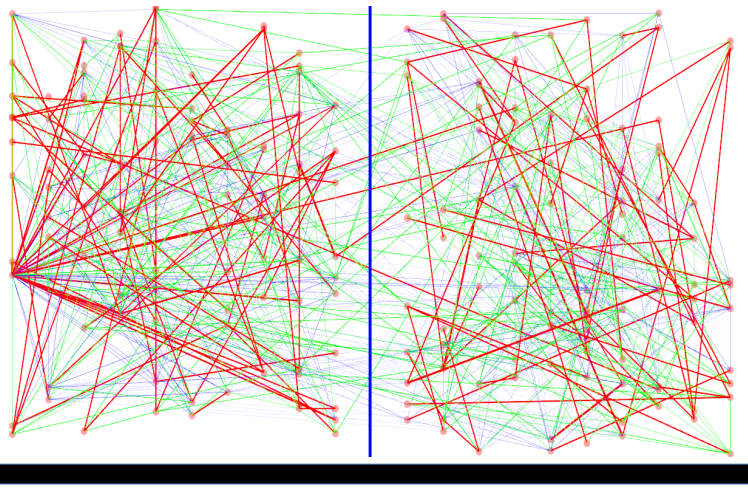
Initial Cut Size	Final Cut Size	Number of passes
157.69	30.58	3

EXPERIMENT NUMBER : 4

Initial random partition:



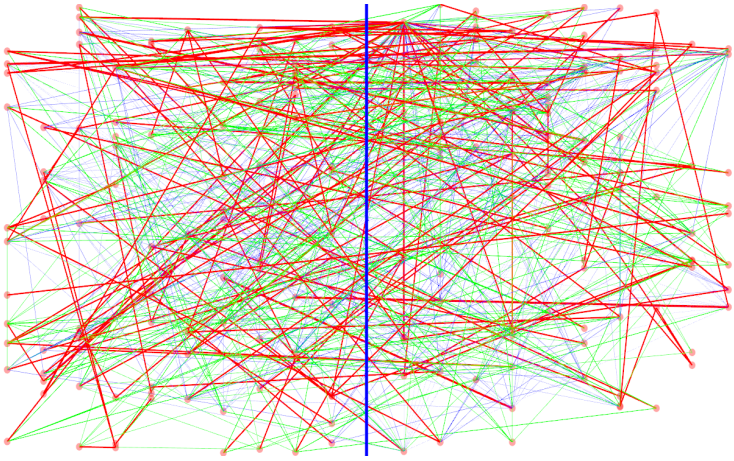
Final partitioining:



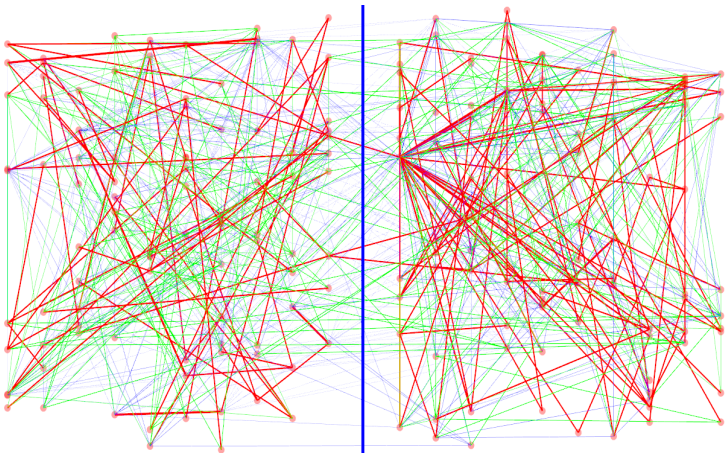
Initial Cut Size	Final Cut Size	Number of passes
155.6	34.56	4

EXPERIMENT NUMBER : 5

Initial random partition:



Final partitioining:



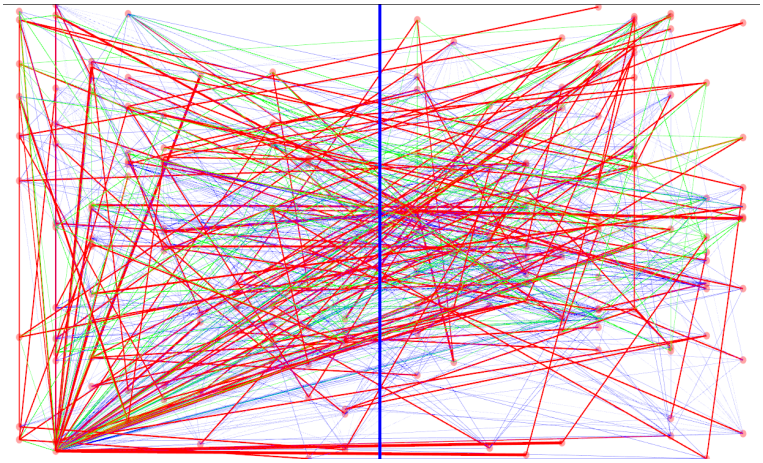
Initial Cut Size	Final Cut Size	Number of passes
171.06	30.58	4

Benchmark circuit 5: ***193 nodes 227 edges***

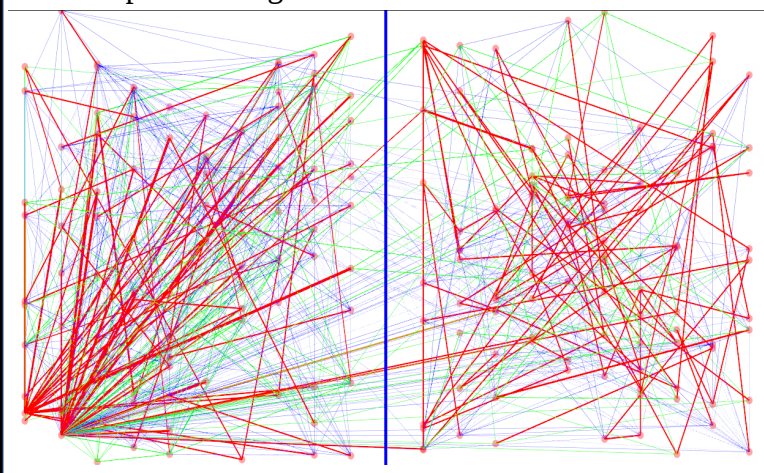
Files: spp_N193_E227_R11_153.nodes.txt
spp_N193_E227_R11_153.nets.txt

EXPERIMENT NUMBER : 1

Initial random partition:



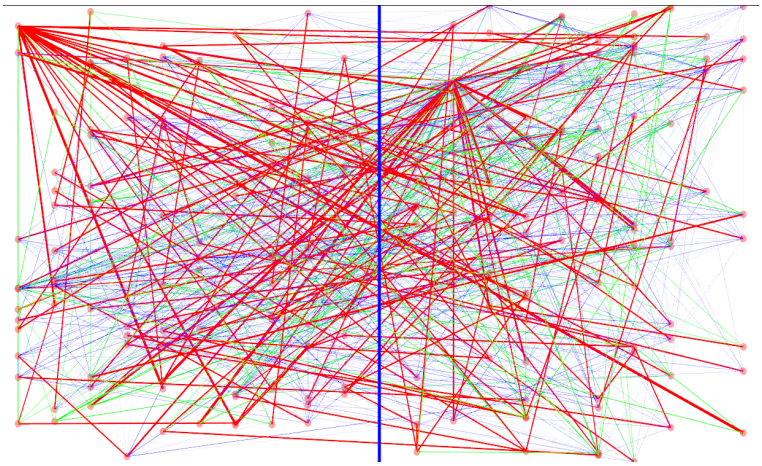
Final partitioining:



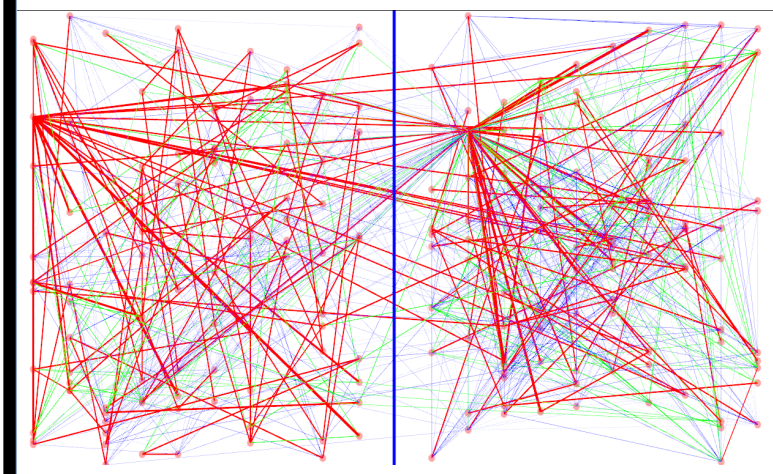
Initial Cut Size	Final Cut Size	Number of passes
169.56	26.96	5

EXPERIMENT NUMBER : 2

Initial random partition:



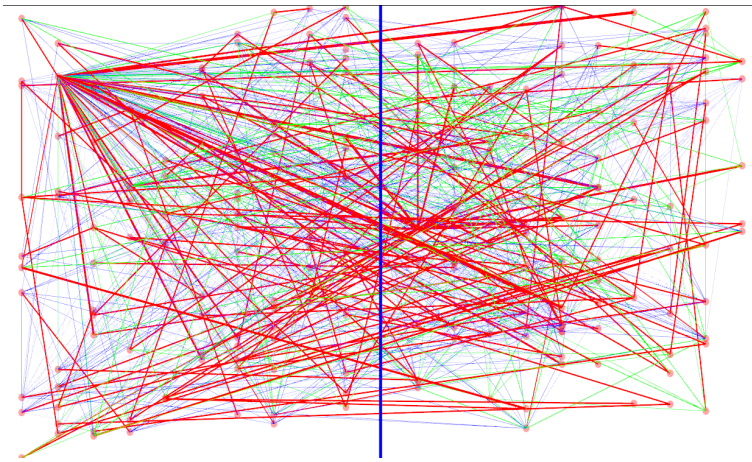
Final partitioining:



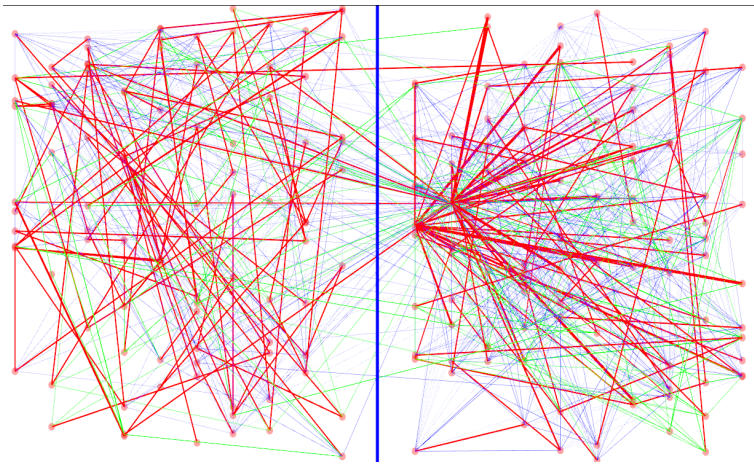
Initial Cut Size	Final Cut Size	Number of passes
159.5	37.69	4

EXPERIMENT NUMBER : 3

Initial random partition:



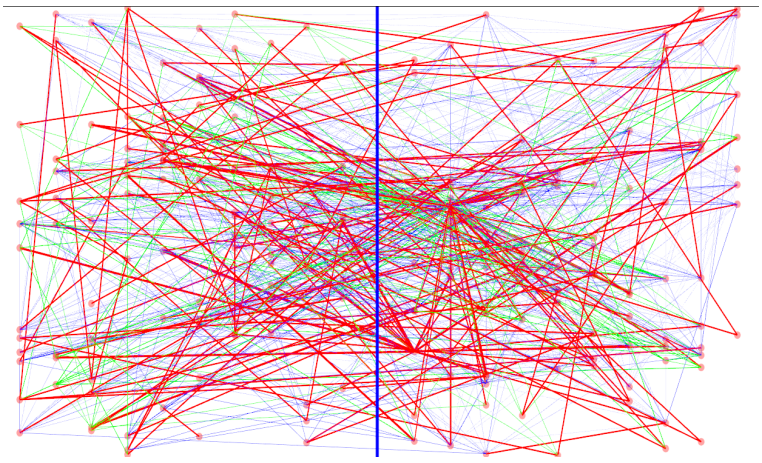
Final partitioining:



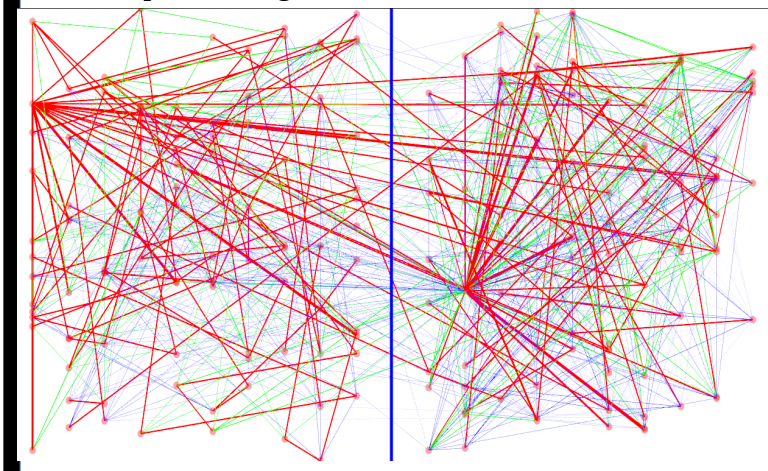
Initial Cut Size	Final Cut Size	Number of passes
170.08	27.96	4

EXPERIMENT NUMBER : 4

Initial random partitioin:



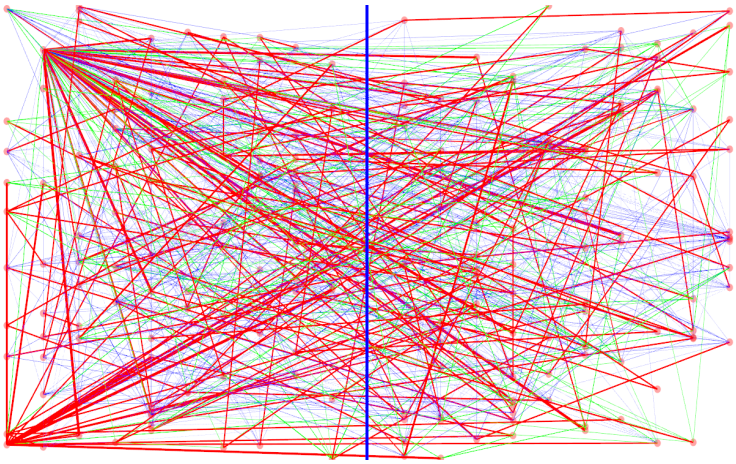
Final partitioning:



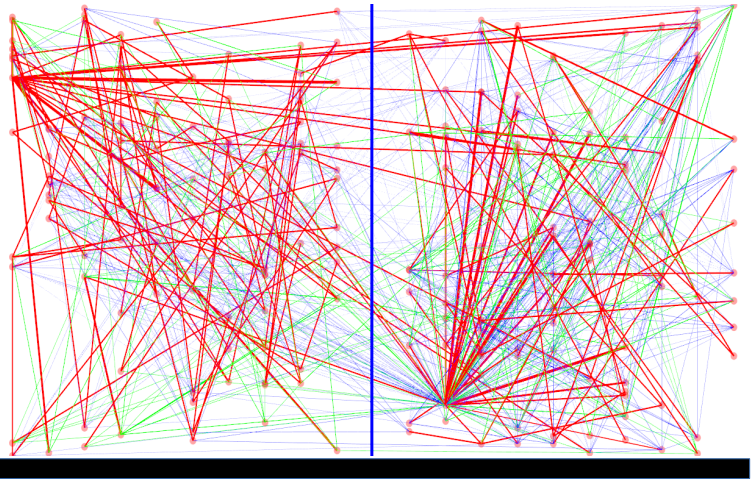
Initial Cut Size	Final Cut Size	Number of passes
160.53	44.7	5

EXPERIMENT NUMBER : 5

Initial random partition:



Final partitioning:



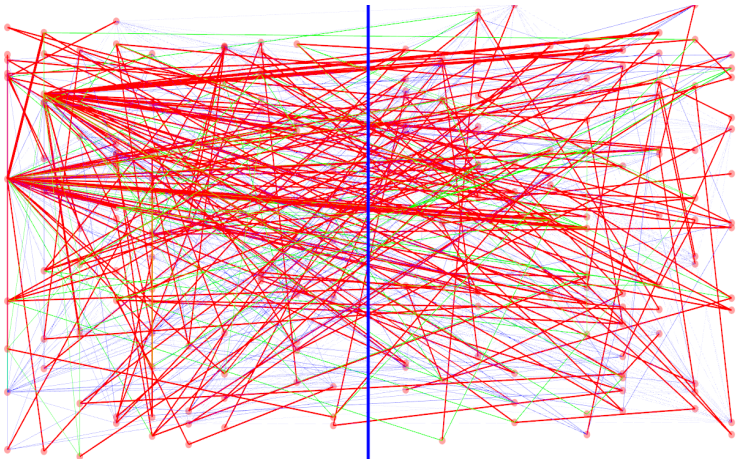
Initial Cut Size	Final Cut Size	Number of passes
175.15	43.41	4

Benchmark circuit 6: ***199 nodes 232 edges***

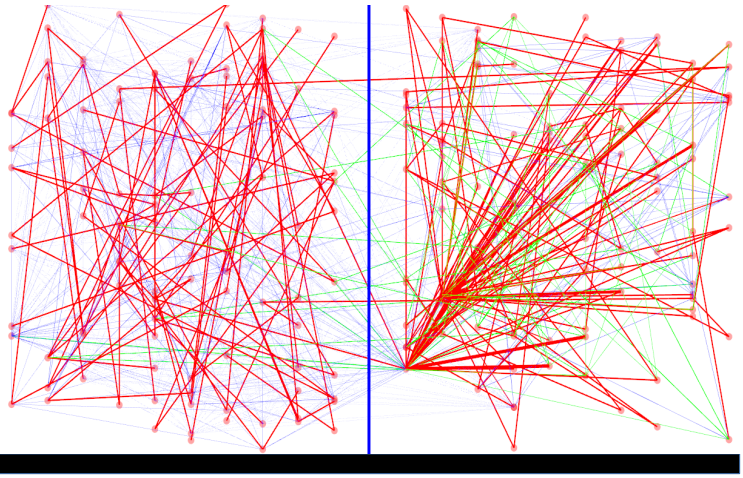
Files: spp_N199_E232_R11_154.nodes.txt
spp_N199_E232_R11_154.nets.txt

EXPERIMENT NUMBER : 1

Initial random partition:



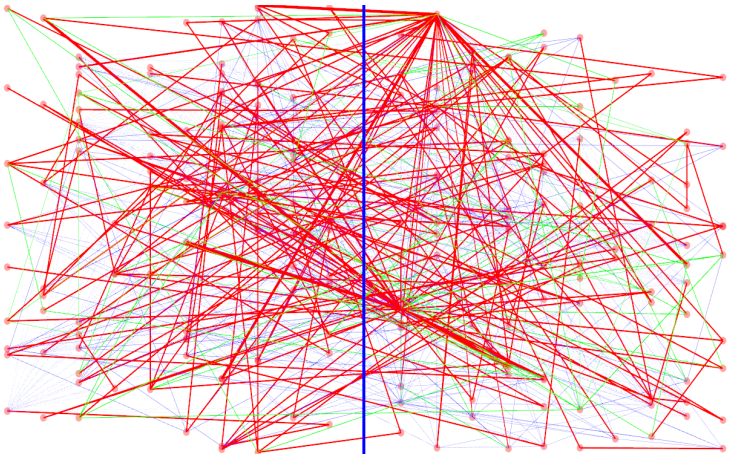
Final partitioning:



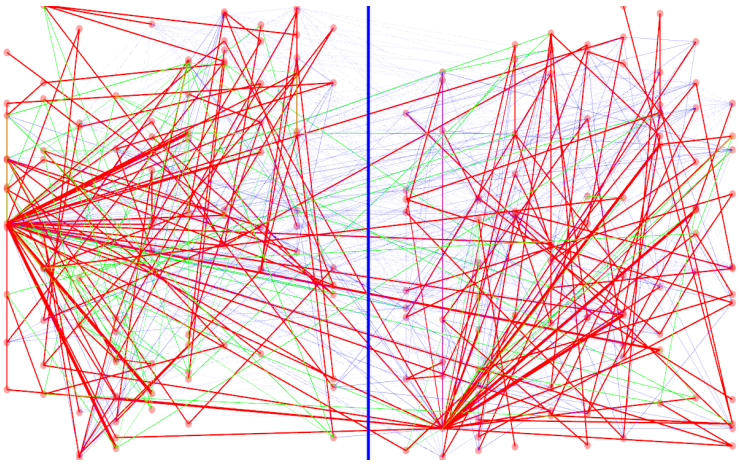
Initial Cut Size	Final Cut Size	Number of passes
147.33	21.58	6

EXPERIMENT NUMBER : 2

Initial random partition:



Final partitioning

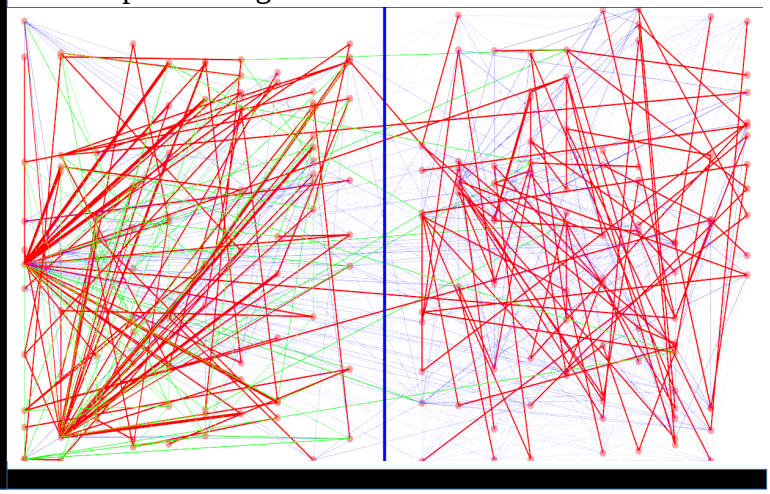


Initial Cut Size	Final Cut Size	Number of passes
143.19	39.94	5

EXPERIMENT NUMBER : 3

Initial random partitioin:

Final partitioning:

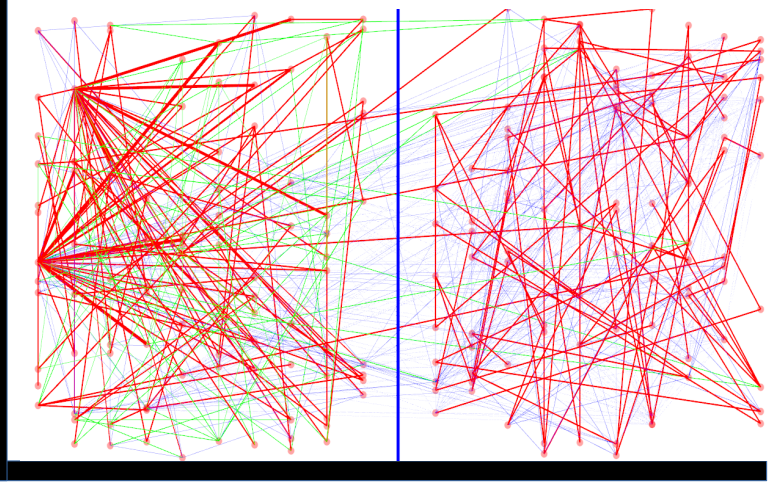


Initial Cut Size	Final Cut Size	Number of passes
141.26	20.4801	7

EXPERIMENT NUMBER : 4

Initial random partitioin:

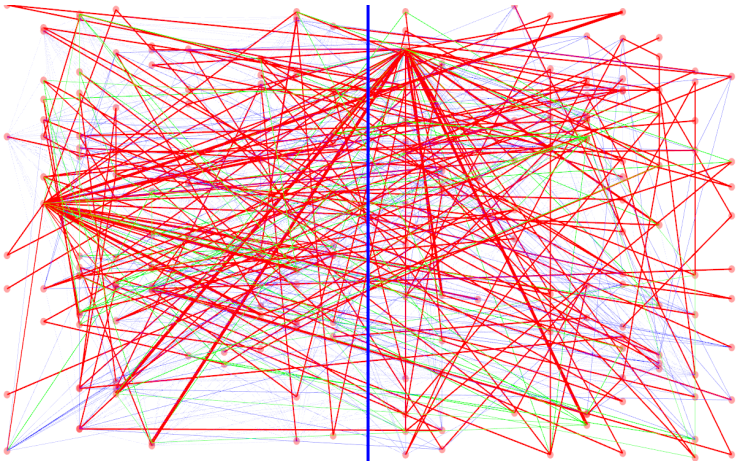
Final partitioning:



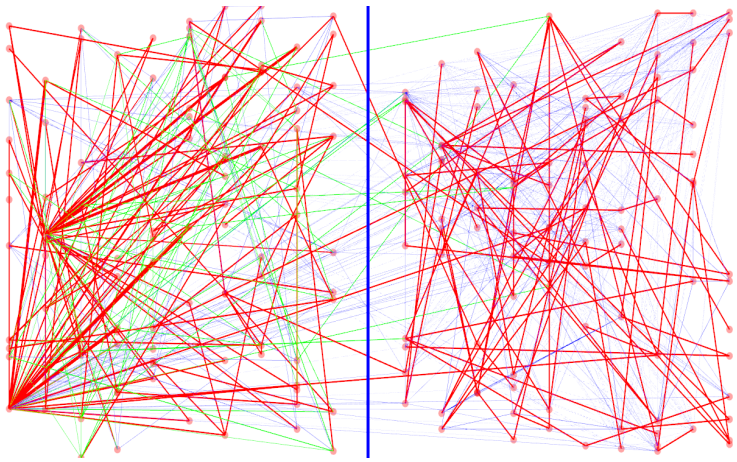
Initial Cut Size	Final Cut Size	Number of passes
142.77	20.48	6

EXPERIMENT NUMBER : 5

Initial random partition:



Final partitioning:



Initial Cut Size	Final Cut Size	Number of passes
132.66	20.56	5

Testing strategy

Following mini circuits were used for testing the implementation before running it on the benchmark circuits:

- 4 nodes hypergraph
  - Files : spp_N4_E2_R11_80_nodes.txt spp_N4_E2_R11_80_nets.txt
- 10 nodes graph
  - Files : spp_N10_E17_R11_80_nodes.txt spp_N10_E17_R11_80_nets.txt

Contents of the four files are:

- cat spp_N4_E2_R11_80_nodes.txt

```
#UCLA nodes 1.0
# Created      : Wed Sep 22 14:45:00 1999
# User         : caldwell@nexus11.cs.ucla.edu (Andrew Caldwell)
# Platform     : SunOS 5.7 sparc SUNW,Ultra-5_10

NumNodes :            4
NumTerminals :        0
a0
a1
a2
a3
```

- cat spp_N4_E2_R11_80_nets.txt

```
#UCLA nodes 1.0
# Created      : Wed Sep 22 14:45:00 1999
# User         : caldwell@nexus11.cs.ucla.edu (Andrew Caldwell)
# Platform     : SunOS 5.7 sparc SUNW,Ultra-5_10

NumPins : 437
NetDegree : 4
a0 B
a1 B
a2 B
a3 B
NetDegree : 2
a2 B
a3 B
```

- cat spp_N10_E17_R11_80_nodes.txt

```
#UCLA nodes 1.0
# Created      : Wed Sep 22 14:45:00 1999
# User         : caldwell@nexus11.cs.ucla.edu (Andrew Caldwell)
# Platform     : SunOS 5.7 sparc SUNW,Ultra-5_10

NumNodes :            10
NumTerminals :        0
a0
a1
a2
a3
a4
a5
a6
a7
a8
a9
```

- cat spp_N10_E17_R11_80_nets.txt

```
#UCLA nets 1.0
# Created      : Wed Sep 22 14:45:00 1999
# User         : caldwell@nexus11.cs.ucla.edu (Andrew Caldwell)
# Platform     : SunOS 5.7 sparc SUNW,Ultra-5_10

NumPins : 437
NetDegree : 2
a0 B
a3 B
NetDegree : 2
a0 B
a7 B
NetDegree : 2
a0 B
a9 B
NetDegree : 2
a1 B
a6 B
NetDegree : 2
a1 B
a7 B
NetDegree : 2
a2 B
a5 B
NetDegree : 2
a2 B
a6 B
NetDegree : 2
a2 B
a8 B
NetDegree : 2
a3 B
a5 B
NetDegree : 2
a3 B
a6 B
NetDegree : 2
a3 B
a9 B
NetDegree : 2
a4 B
a5 B
NetDegree : 2
a4 B
a6 B
NetDegree : 2
a4 B
a7 B
NetDegree : 2
a6 B
```

a9 B
NetDegree : 2
a7 B
a8 B
NetDegree : 2
a7 B
a9 B

Future Work

- The same implementation could be implemented using string based data structure where each group is saved as a string, the node is represented as an integer number in the group string. These integer numbers can be used to index into the array of nodes. Neighboring nodes can be saved as string of integer nodes, the strings being stored in an array of string nodes where each array element number represents a node.
- Intelligent initial random partitioning whereby the nodes are partitioned such that the number of passes for the KL algorithm are decreased.

References

* http://cs.baylor.edu/~maurer/partitioning.pdf

* https://en.wikipedia.org/wiki/Graph_partition

* https://en.wikipedia.org/wiki/Kernighan%E2%80%93Lin_algorithm

* https://github.com/abangfarhan/graph-sfml

* Lecture Notes: Physical Design Automation for VLSI and FPGAs: Lecture 3: Partitioning

---